Beej's Guide to Unix IPC

Brian “Beej Jorgensen” Hall
beej@beej.us

Version 1.1.3
December 1, 2015

11. Unix Sockets

Remember FIFOs? Remember how they can only send data in one direction, just like a Pipes? Wouldn't it be grand if you could send data in both directions like you can with a socket?

Well, hope no longer, because the answer is here: Unix Domain Sockets! In case you're still wondering what a socket is, well, it's a two-way communications pipe, which can be used to communicate in a wide variety of *domains*. One of the most common domains sockets communicate over is the Internet, but we won't discuss that here. We will, however, be talking about sockets in the Unix domain; that is, sockets that can be used between processes on the same Unix system.

Unix sockets use many of the same function calls that Internet sockets do, and I won't be describing all of the calls I use in detail within this document. If the description of a certain call is too vague (or if you just want to learn more about Internet sockets anyway), I arbitrarily suggest *Beej's Guide to Network Programming using Internet Sockets*[41]. I know the author personally.

11.1. Overview

Like I said before, Unix sockets are just like two-way FIFOs. However, all data communication will be taking place through the sockets interface, instead of through the file interface. Although Unix sockets are a special file in the file system (just like FIFOs), you won't be using **`open()`** and **`read()`**—you'll be using **`socket()`**, **`bind()`**, **`recv()`**, etc.

When programming with sockets, you'll usually create server and client programs. The server will sit listening for incoming connections from clients and handle them. This is very similar to the situation that exists with Internet sockets, but with some fine differences.

For instance, when describing which Unix socket you want to use (that is, the path to the special file that is the socket), you use a `struct sockaddr_un`, which has the following fields:

```
struct sockaddr_un {
    unsigned short sun_family;  /* AF_UNIX */
    char sun_path[108];
}
```

This is the structure you will be passing to the **`bind()`** function, which associates a socket descriptor (a file descriptor) with a certain file (the name for which is in the *`sun_path`* field).

11.2. What to do to be a Server

Without going into too much detail, I'll outline the steps a server program usually has to go through to do it's thing. While I'm at it, I'll be trying to implement an “echo server” which just echos back everything it gets on the socket.

Here are the server steps:

1. **Call `socket()`:** A call to **`socket()`** with the proper arguments creates the Unix socket:

   ```
   unsigned int s, s2;
   struct sockaddr_un local, remote;
   int len;

   s = socket(AF_UNIX, SOCK_STREAM, 0);
   ```

   The second argument, `SOCK_STREAM`, tells **`socket()`** to create a stream socket. Yes, datagram sockets (`SOCK_DGRAM`) are supported in the Unix domain, but I'm only going to cover stream sockets here. For the curious, see Beej's Guide to Network Programming[42] for a good description of unconnected datagram sockets that applies perfectly well to Unix sockets. The only thing that changes is that you're now using a `struct sockaddr_`**`un`** instead of a `struct sockaddr_`**`in`**.

   One more note: all these calls return `-1` on error and set the global variable *`errno`* to reflect whatever went wrong. Be sure to do your error checking.

41. `http://beej.us/guide/bgnet/`
42. `http://beej.us/guide/bgnet/`

2. **Call `bind()`:** You got a socket descriptor from the call to **`socket()`**, now you want to bind that to an address in the Unix domain. (That address, as I said before, is a special file on disk.)

```
local.sun_family = AF_UNIX;  /* local is declared before socket() ^ */
strcpy(local.sun_path, "/home/beej/mysocket");
unlink(local.sun_path);
len = strlen(local.sun_path) + sizeof(local.sun_family);
bind(s, (struct sockaddr *)&local, len);
```

   This associates the socket descriptor "*s*" with the Unix socket address "`/home/beej/mysocket`". Notice that we called **`unlink()`** before **`bind()`** to remove the socket if it already exists. You will get an `EINVAL` error if the file is already there.

3. **Call `listen()`:** This instructs the socket to listen for incoming connections from client programs:

```
listen(s, 5);
```

   The second argument, `5`, is the number of incoming connections that can be queued before you call **`accept()`**, below. If there are this many connections waiting to be accepted, additional clients will generate the error `ECONNREFUSED`.

4. **Call `accept()`:** This will accept a connection from a client. This function returns *another socket descriptor*! The old descriptor is still listening for new connections, but this new one is connected to the client:

```
len = sizeof(struct sockaddr_un);
s2 = accept(s, &remote, &len);
```

   When **`accept()`** returns, the *`remote`* variable will be filled with the remote side's `struct sockaddr_un`, and *`len`* will be set to its length. The descriptor *`s2`* is connected to the client, and is ready for **`send()`** and **`recv()`**, as described in the Network Programming Guide[43].

5. **Handle the connection and loop back to `accept()`:** Usually you'll want to communicate to the client here (we'll just echo back everything it sends us), close the connection, then **`accept()`** a new one.

```
while (len = recv(s2, &buf, 100, 0), len > 0)
    send(s2, &buf, len, 0);

/* loop back to accept() from here */
```

6. **Close the connection:** You can close the connection either by calling `close()`, or by calling **`shutdown()`**[44].

With all that said, here is some source for an echoing server, *`echos.c`*[45]. All it does is wait for a connection on a Unix socket (named, in this case, "echo_socket").

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, s2, t, len;
    struct sockaddr_un local, remote;
    char str[100];
```

43. `http://beej.us/guide/bgnet/`
44. `http://beej.us/guide/url/shutdownman`
45. `http://beej.us/guide/bgipc/examples/echos.c`

```
    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    local.sun_family = AF_UNIX;
    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    len = strlen(local.sun_path) + sizeof(local.sun_family);
    if (bind(s, (struct sockaddr *)&local, len) == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(s, 5) == -1) {
        perror("listen");
        exit(1);
    }

    for(;;) {
        int done, n;
        printf("Waiting for a connection...\n");
        t = sizeof(remote);
        if ((s2 = accept(s, (struct sockaddr *)&remote, &t)) == -1) {
            perror("accept");
            exit(1);
        }

        printf("Connected.\n");

        done = 0;
        do {
            n = recv(s2, str, 100, 0);
            if (n <= 0) {
                if (n < 0) perror("recv");
                done = 1;
            }

            if (!done)
                if (send(s2, str, n, 0) < 0) {
                    perror("send");
                    done = 1;
                }
        } while (!done);

        close(s2);
    }

    return 0;
}
```

As you can see, all the aforementioned steps are included in this program: call **`socket()`**, call **`bind()`**, call **`listen()`**, call **`accept()`**, and do some network **`send()`**s and **`recv()`**s.

11.3. What to do to be a client

There needs to be a program to talk to the above server, right? Except with the client, it's a lot easier because you don't have to do any pesky **`listen()`**ing or **`accept()`**ing. Here are the steps:

1. Call **`socket()`** to get a Unix domain socket to communicate through.
2. Set up a `struct sockaddr_un` with the remote address (where the server is listening) and call **`connect()`** with that as an argument
3. Assuming no errors, you're connected to the remote side! Use **`send()`** and **`recv()`** to your heart's content!

How about code to talk to the echo server, above? No sweat, friends, here is *echoc.c*[46]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, t, len;
    struct sockaddr_un remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    printf("Trying to connect...\n");

    remote.sun_family = AF_UNIX;
    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);
    if (connect(s, (struct sockaddr *)&remote, len) == -1) {
        perror("connect");
        exit(1);
    }

    printf("Connected.\n");

    while(printf("> "), fgets(str, 100, stdin), !feof(stdin)) {
        if (send(s, str, strlen(str), 0) == -1) {
            perror("send");
            exit(1);
        }

        if ((t=recv(s, str, 100, 0)) > 0) {
            str[t] = '\0';
            printf("echo> %s", str);
        } else {
            if (t < 0) perror("recv");
            else printf("Server closed connection\n");
            exit(1);
        }
    }

    close(s);

    return 0;
}
```

In the client code, of course you'll notice that there are only a few system calls used to set things up: **`socket()`** and **`connect()`**. Since the client isn't going to be **`accept()`**ing any incoming connections, there's no need for it to **`listen()`**. Of course, the client still uses **`send()`** and **`recv()`** for transferring data. That about sums it up.

46. `http://beej.us/guide/bgipc/examples/echoc.c`

11.4. `socketpair()`—quick full-duplex pipes

What if you wanted a **`pipe()`**, but you wanted to use a single pipe to send and recieve data from *both sides*? Since pipes are unidirectional (with exceptions in SYSV), you can't do it! There is a solution, though: use a Unix domain socket, since they can handle bi-directional data.

What a pain, though! Setting up all that code with **`listen()`** and **`connect()`** and all that just to pass data both ways! But guess what! You don't have to!

That's right, there's a beauty of a system call known as **`socketpair()`** this is nice enough to return to you a pair of *already connected sockets*! No extra work is needed on your part; you can immediately use these socket descriptors for interprocess communication.

For instance, lets set up two processes. The first sends a `char` to the second, and the second changes the character to uppercase and returns it. Here is some simple code to do just that, called *`spair.c`*[47] (with no error checking for clarity):

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
    int sv[2]; /* the pair of socket descriptors */
    char buf; /* for data exchange between processes */

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) == -1) {
        perror("socketpair");
        exit(1);
    }

    if (!fork()) {  /* child */
        read(sv[1], &buf, 1);
        printf("child: read '%c'\n", buf);
        buf = toupper(buf);  /* make it uppercase */
        write(sv[1], &buf, 1);
        printf("child: sent '%c'\n", buf);

    } else { /* parent */
        write(sv[0], "b", 1);
        printf("parent: sent 'b'\n");
        read(sv[0], &buf, 1);
        printf("parent: read '%c'\n", buf);
        wait(NULL); /* wait for child to die */
    }

    return 0;
}
```

Sure, it's an expensive way to change a character to uppercase, but it's the fact that you have simple communication going on here that really matters.

One more thing to notice is that **`socketpair()`** takes both a domain (`AF_UNIX`) and socket type (`SOCK_STREAM`). These can be any legal values at all, depending on which routines in the kernel you want to handle your code, and whether you want stream or datagram sockets. I chose `AF_UNIX` sockets because this is a Unix sockets document and they're a bit faster than `AF_INET` sockets, I hear.

Finally, you might be curious as to why I'm using **`write()`** and **`read()`** instead of **`send()`** and **`recv()`**. Well, in short, I was being lazy. See, by using these system calls, I don't have to enter the *`flags`* argument that **`send()`** and **`recv()`** use, and I always set it to zero anyway. Of course, socket descriptors are just file descriptors like any other, so they respond just fine to many file manipulation system calls.

47. http://beej.us/guide/bgipc/examples/spair.c